Working with a computer proof assistant

Andreas Mair & Martin Fuchs

1st February 2022

1 Introduction

Checking the validity of theorems in mathematics relies on a chain of precisely formulated arguments what everybody refers to as proofs. Formally verifying every link in this chain may at times involve tedious calculations or the use of other already proven theorems, lemmas etc. Computer proof assistants lend a helping hand as formal verification can in part be automated or supported by an interactive computer program.

In this paper we are going to provide an introduction to Lean, a theorem proof assistant, and to discover the possibilities of this computer proof assistant using examples from the Natural Number Game. The game lets players prove fundamental theorems about the natural numbers such as the commutativity of addition from the ground up and we will soon realise that interactive theorem provers such as Lean provide great assistance in proving statements with many dependencies and finicky axiomatic details. Furthermore, we will take a look at the link between Lean and category theory by giving a short introduction to Cartesian closed categories. In essence, we need to allow morphisms as objects in our category giving us terms in Lean that are of the type *morphism*.

2 The Lean theorem prover

Lean is a functional programming language invented by Leonardo de Moura at Microsoft Research in 2013. The language can be used as an interactive theorem prover meaning that mathematical statements that are formulated in a type theoretic language can be proven using so called tactics. There have been attempts to prove a wide variety of theorems and lemmas like the Sylow theorems or the intermediate value theorem. Notably, Kevin Buzzard Professor of Mathematics at Imperial College London initiated the Xena project whose goal it is to formalise and prove every theorem in the undergraduate maths curriculum at his university.

An advantage of Lean is the type theoretic basis of functional programming. Every object has a type, for example

```
constant m : nat
```

where the constant variable `m` now has the type *natural number*. Colons signify the relationship `term : type`. What makes Lean so powerful is the definition of types that are themselves functions from one type to another. For example

```
constant f : nat → nat
```

is now a term that has the type of an endomorphism on the natural numbers. We can readily repeat this process under the convention that Lean always starts putting in brackets from the right, i.e.

```
Type → Type → Type = Type → (Type → Type).
```

Types like `nat`, the natural numbers, or `bool`, the booleans, can themselves be instantiated on a sort of meta level. For this Lean introduces the type `Type` and one can define objects like

```
constants α β : Type
constant F : Type → Type
constant G : Type → Type → Type
```

For example F is now an object of a type that maps some type to another type. The functions can be applied to arguments as expected. Moreover, Lean also allows for function abstraction

```
constants α β : Type
constant f : α → β
#check λ x : α, f x

Returns:
λ (x : α), f x : α → β
```

which is like saying that the term is a function that maps $x \mapsto f(x)$. This is a very powerful tool in functional programming that exhibits a neat connection to category theory. The symbol λ is a notation that stems from the lambda calculus, a formal language to investigate and express functions with that was developed in the 1930s by Alonzo Church at Princeton University. We will get back to this in Section 4.

Moreover, Lean supports the use of tactics. In Lean, proofs or more precisely representations of mathematical proofs are built using tactics. In a sense, they are commands or instructions on *how to* prove a statement. This is like formalising the recipe

> "to prove the forward direction, **unfold** the definition, **apply** the previous lemma and **simplify**".

In the next chapter we will see common tactics as they come up in the natural number game.

3 The Natural Number Game

The Natural Number Game is an educational web game made by Kevin Buzzard and Mohammad Pedramfar. It teaches about using computer proof assistants and the mathematical power of induction. One is using a slightly altered online version of Lean 3 to prove some basic facts about the natural numbers. It has currently got ten chapters with several levels in each chapter. You start off with a set of the natural numbers called `mynat`, Peano's axioms and the principle of mathematical induction. In each level you try to prove a small theorem about the natural numbers only using the facts you have proven in the levels before. Lean calls theorems and statements that need to be proven goals. The game introduces you to several of the tactics of Lean along the way, some of which we will introduce below.

3.1 Peano's axoims: mynat.definition

For a start we need to put forward Peano's axioms which characterise the natural numbers.

Definition 3.1 (Peano's axioms)**.** Let $\mathbb{N}$ denote the set of natural numbers defined by the following five axioms.

(*i*) 0 is a natural number.

(*ii*) Every natural number n has a successor $\text{succ}(n)$.

(*iii*) 0 is not a successor of any other natural number.

(*iv*) Natural numbers that have the same successor are equal.

(*v*) If a set X contains 0 and for every natural number n its successor $\text{succ}(n)$ then $\mathbb{N} \subseteq X$.

Additionally, we have the addition and multiplication on the natural numbers which are defined inductively as follows.

Definition 3.2 (Addition and multiplication)**.** Let $m, n \in \mathbb{N}$.
For **addition**:

(*i*) $m + 0 := m$

(*ii*) $m + \text{succ}(n) := \text{succ}(m + n)$

And for **multiplication**:

(*i*) $m \cdot 0 := 0$

(*ii*) $m \cdot \text{succ}(n) := (m \cdot n) + m$

For the definition of the natural numbers in Lean this boils down to

- a term `0 : mynat`,
- a function `succ : mynat → mynat` with `succ n` interpreted as the successor of `n`.
- and the principle of mathematical induction.

Furthermore, we get four theorems about addition and multiplication that are axiomatically true.

```
add_zero (a : mynat) : a + 0 = a
add_succ (a b : mynat) : a + succ(b) = succ(a + b)

mul_zero (a : mynat) : a * 0 = 0
mul_succ (a b : mynat) : a * succ(b) = a * b + a
```

3.2 The first steps: rw, refl

The natural number game starts off with a very gentle introduction. Often times proofs rely on substituting terms and checking that two sides of an equation are equivalent, i.e. definitionally the same. Let us start with a simple lemma

Lemma 3.1. *For all* $a \in \mathbb{N}$

$$a + \mathrm{succ}(0) = \mathrm{succ}(a).$$

For which we create a goal in Lean by typing

```
lemma add_succ_zero (a : mynat) : a + succ(0) = succ(a) :=
```

Here the lemma `add_succ_zero` is a proof of the type `a + succ(0) = succ(a)` and we need to come up with a "recipe" on how to construct a term of this type. From the inductive definition of addition we know how to add a successor to a natural number and we know that $n + 0 = n$. The first tactic we thus use is `rw` which substitutes part of an expression by something that is definitionally equal. For example

```
rw add_succ,
rw add_zero,
```

first changes the goal `a + succ(0) = succ(a)` to `succ(a + 0) = succ(a)` since `n + succ(m) = succ(n + m)` for all $n, m \in \mathbb{N}$ and then replaces `a + 0` by `a` resulting in a goal `succ(a) = succ(a)`. In a sense we are almost done since the two sides of the equation are the same expression. We tell Lean to check if they are the same and close the goal if so by the `refl` tactic. Thus

```
rw add_succ,
rw add_zero,
refl,
```

is a proof of the lemma.

3.3 induction

From the the fifth Peano axiom we get the principle of mathematical induction. Say we want to prove the following simple lemma

Lemma 3.2. *For all* $n \in \mathbb{N}$

$$0 + n = n.$$

Intuitively, we would proof this statement inductively. In Lean the `induction` tactic facilitates this step. First we create the goal:

```
lemma zero_add (n : mynat) : 0 + n = n :=
```

Then we introduce the induction:

```
induction n with d hd,
```

The line above creates two separate goals we are already familiar with, namely, the base case and the induction step where `hd` is the induction hypothesis and `d : mynat`. This means that first need to prove $0 + 0 = 0$ by

```
rw add_zero,
refl,
```

and then use the second axiom of addition to prove the induction step by

```
rw add_succ,
rw hd,
refl,
```

Remember that after `rw add_succ` our goal is to prove that $\text{succ}(0+d) = \text{succ}(d)$ which is true by the induction hypothesis $0+d=d$. So, `rw hd` uses the induction hypothesis and `refl` finalises the proof.

3.4 intro, apply

We can think of implications like functions. For example, if we want to prove some statement Q and we know that from some other statement P that Q always follows we can apply the "function" from P to Q to any proof of statement P. In Lean this procedure is realised by the `apply` tactic. If our goal is to prove an implication $P \implies Q$, then we want to pick a proof of P, i.e. a term p of type P and describe how to map it to a proof q of Q. This introduction of a proof p of P is achieved by `intro` We now want to take a look at a proof of the following statement.

Theorem 3.1 (Right cancellation property)**.** *For natural numbers $a, b, t \in N$*

$$a + t = b + t \implies a = b.$$

This is called the right cancellation property of addition.

Take note that by the fourth Peano axiom we see that the successor function is injective a fact that can be accessed in Lean via `succ_inj`. We begin the proof by setting the goal.

```
theorem add_right_cancel (a t b : mynat) : a + t = b + t → a = b :=
```

Let us introduce a term of type `a + t = b + t` and use induction over t

```
intro h
induction t with d hd,
```

The base case is simple. From $a+0=b+0$ follows $a=b$ by `add_zero`. So, we are left with the induction step where by the hypothesis $a+\text{succ}(d) = b+\text{succ}(d)$ and by the induction hypothesis $a+d=b+d \implies a=b$ hold true. Let us combine these two hypotheses with

```
rw add_succ at h,
rw add_succ at h,
```

Which turns the hypothesis `h : a + succ d = b + succ d` into `h : succ (a + d) = succ (b + d)`. Now we use `apply` for the first time. The induction hypothesis is an implication, specifically `hd : a + d = b + d → a = b`. So we can use this to turn the goal of constructing a term of type `a = b` into a goal of constructing a term of type `a + d = b + d`. Then, because of the successor injectivity we again use `apply` to turn the goal of constructing a term of type `a + d = b + d` into a goal of constructing a term of type `succ (a + d) = succ (b + d)`. But this is exactly what our hypothesis is which finishes the proof. In code this means

```
apply hd,
apply succ_inj,
exact h,
```

where `exact h` tells Lean that the term h is of the type we sought after.

3.5 cases, exfalso

In this section we are going to prove the following theorem.

Theorem 3.2. *For all $n, m \in \mathbb{N}$ non-zero*

$$n \cdot m \neq 0.$$

First, we need to talk about the way Lean represents a "not" statement, e.g. $m \neq n$. The concept can be turned into a function such that a proof of the statement

```
p : m ≠ n
```

turns into

```
p : (m = n) → false
```

This intuitively makes sense. Since m and n are not equal we can see a proof of this fact as a map that expresses the falsehood of every statement that says otherwise. With that out of the way let us introduce `exfalso`.

This tactic can be used to prove a statement by contraposition. Imagine our goal is to prove a false statement. How do we come up with a proof? Well, we do not. We use the `exfalso` tactic and rather try to construct a map that maps to the type `false`. We will see shortly how this works. Before, let us talk about the `cases` tactic. Often times in mathematics one is required to check a statement on a case by case basis. Lean provides us with such a tool as well. Invoking the tactic splits a goal into two subgoals. First, one proves the case where the variable at hand is equal to zero and second, one proves the statement where the variable is a successor of some other natural number. Let us see how this works and define the theorem

```
theorem mul_pos (a b : mynat) : a ≠ 0 → b ≠ 0 → a * b ≠ 0 :=
```

We introduce terms `ha, hb, hab` of the types `a ≠ 0`, `b ≠ 0` and `a * b = 0` respectively by

```
intros ha, hb, hab,
```

Remember that the last type is an equality since the type `a * b ≠ 0` definitionally equals `(a * b = 0) → false` for Lean which by use of `intro` means picking an element in the domain, i.e. `hab : a * b = 0`. We continue by checking some cases.

```
cases b with b,
apply hb,
refl,
```

In the case where $b = 0$ we can use `hb : b ≠ 0` meaning `(b = 0) → false` since $0 \neq 0$ actually is a false statement. `refl` closes the goal for this case. Now we turn our attention to the case where the variable is succ(b). Then

```
rw mul_succ at hab,
apply ha,
cases a with a,
refl,
```

where we have cleaned up $a * \text{succ}(b) = 0$ into $a * b + a = 0$ used $a \neq 0$ to go from a goal of type `false` to a goal of type `a = 0` and opened a new case analysis for the variable a. `refl` directly closes the first subgoal for $a = 0$ and then we finalise the proof by

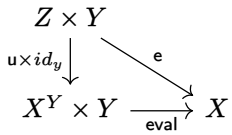
```
rw add_succ at hab,
exfalso,
apply succ_ne_zero (succ a * b + a)
exact hab,
```

In this case we look at the successor $\mathrm{succ}(a)$. Thus first, we clean up the hypothesis $\mathrm{succ}(a) * b + \mathrm{succ}(a) = 0$ and get $\mathrm{succ}((\mathrm{succ}(a) * b + a)) = 0$. Our current goal is to construct something of type $\mathrm{succ}(c) = 0$ for some `c : mynat` which is evidently a false statement since no successor is equal to zero. Therefore, we invoked `exfalso` and finished the proof by using the the simplified hypothesis.

4 Cartesian closed categories

Before we are able to define the notion of a Cartesian closed category we need to introduce the concept of exponential objects.

Definition 4.1 (Exponential object)**.** Let X and Y be objects of a category $\mathbf{C}$ such that all binary products with Y exist. Then an *exponential object* is an object X^Y equipped with an evaluation map $\mathsf{eval} : X^Y \times Y \to X$ which is universal in the sense that, given any object Z and map $\mathsf{e} : Z \times Y \to X$, there exists a unique map $\mathsf{u} : Z \to X^Y$ such that



commutes.

Remark/Example 4.1. (i) An equivalent way to define the exponential object is in terms of adjoints. We require the product functor

$$\begin{aligned} - \times Y : \mathbf{C} &\to \mathbf{C} \\ X &\mapsto X \times Y \end{aligned}$$

to have a right adjoint. Suppose the functor F is the right adjoint of the product functor then naturally

$$\mathbf{C}(Z, FX) \cong \mathbf{C}(Z \times Y, X).$$

If we take Z = FX, then

$$\mathbf{C}(FX, FX) \cong \mathbf{C}(FX \times Y, X).$$

The identity morphism thus maps to a morphism $\mathsf{eval} : FX \times Y \to X$ and we call FX the exponential object X^Y where the morphism is the evaluation morphism eval as in the definition above.
(ii) In the category **Set** of sets for $X, Y \in \mathbf{Set}_0$ the exponential object X^Y is the set of function from Y to X. Remember that for the category of finite sets **FinSet** it holds true that

$$|X^Y| = |X|^{|Y|}.$$

(iii) For **Top** the exponential object X^Y for $X, Y \in \mathbf{Top}_0$ exists if Y is a locally compact Hausdorff space. Then X^Y is the set of continuous functions from Y to X together with the compact-open topology, i.e. the topology generated by the subbasis B which constists of sets of continuous functions f from Y to X satisfying the following property. For a compact subset $K \subseteq Y$ and an open subset $U \subseteq X$ it must hold that $f(K) \subseteq U$.

(*iv*) In functional programming (such as when programming in Lean) the morphism eval is often called `apply`.

We may now define Cartesian closed categories and we will shortly see the similarity to functional programming and type theory.

Definition 4.2 (Cartesian closed category)**.** A category $\mathbf{C}$ is called Cartesian closed if

(*i*) it contains a terminal object,

(*ii*) for any two objects X, Y the product $X \times Y$ is an object

(*iii*) and for any two objects X, Y it contains the exponential object X^Y.

Remark/Example 4.2. (*i*) The first two properties may be combined into a single one, namely that $\mathbf{C}$ must contain the product of any finite (possibly empty) family of objects. Then the empty product is a terminal object.
(*ii*) The category **Set** is Cartesian closed. The product $X \times Y$ is the Cartesian product of X and Y. Any singleton set is a terminal object of **Set** and the exponential objects are given as in Remark/Example 4.1.
(*iii*) In **Cat**, the category of locally small categories, the exponential object is given by the functor category, the product by the categorical product and the terminal object is the terminal category, i.e. the category with one object and a single identity morphism.

Having introduced Lean and Cartesian closed categories we can draw the connection between the two.

Remark/Example 4.3 (Connection between CCC and Lean). We have already implicitly used the simply typed calculus in earlier sections. *Simply typed* refers to the existence of types and there only being one way to "generate" new types, namely, by building function types. Interestingly, we can view the simply typed lambda calculus as a Cartesian closed category where

(*i*) objects correspond to types,

(*ii*) morphisms to function types,

(*iii*) the terminal element corresponds to the unit type, i.e. the type with only one element, together with the constant mappings from any other type into it,

(*iv*) for a pair of types A, B the product is just the *product type* of ordered pairs and

(*v*) the exponential object A^B for types A, B is just given as the type of functions from B to A.

Thus, the simply typed lambda calculus can be thought of as the language of the structure of Cartesian closed categories.

References

[1] Kevin Buzzard and Mohammad Pedramfar. *The Natural Number Game*. `https://www.ma.imperial.ac.uk/~buzzard/xena/natural_number_game/`. [Online – accessed 2022-01-19].

[2] Leonardo de Moura Jeremy Avigad and Soonho Kong. *Theorem Proving in Lean*. `https://leanprover.github.io/theorem_proving_in_lean/`. [Online – accessed 2022-01-19].

[3] Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media, 2013.

[4] Paolo Perrone. *Notes on Category Theory with examples from basic mathematics*. 2019. eprint: `arXiv:1912.10642`.